The ITB Journal

Volume 1 | Issue 2 Article 2

2000

Non Deterministic Processing in Neural Networks : An Introduction to Multi-Threaded Neural Networks

Stephen Sheridan

Non Deterministic Processing in Neural Networks

An Introduction to Multi-Threaded Neural Networks

Stephen Sheridan

School of Informatics and Engineering, ITB

Abstract

Since McCullough and Pitts first published their work on the Binary Decision Neuron much research has been accumulated in the area of neural networks. This work has for the most part centred on network topologies and learning algorithms. The neural networks that have found their way into devices such as handheld PC's are the fruit of NN research that has spanned 57 years. There is a simplistic beauty in the way that artificial neural networks model the biological foundations of the human thought process, but one piece of the jigsaw puzzle is still missing. We have so far been unable to match the massive parallelness of the human brain. This paper attempts to explain how multithreaded neural networks can be used as a basis for building parallel networks. By studying simple concurrent networks is hoped that significant inroads can be made into a better understanding of how neural network processing can be spread across multiple processors. The paper outlines some biological foundations and introduces some approaches that may be used to recreate software implementations of concurrent artificial neural networks.

1. Biological foundations

The human brain has been at the centre of much argument since Aristotle's time to the present. Arguments aside one has to marvel at the speed at which the human brain operates. Imagine for one moment you were required to build a robot capable of driving an ordinary everyday car for a distance of 100 meters avoiding some obstacles along the way. Now imagine the thousands of computations the robots artificial brain would have to make for every split second of the cars movement. This is most definitely a huge task for our Robot's artificial brain, but what about the human brain? How can the human brain deal with situations like the one explained above with apparent ease while simultaneously having a conversation with one of the cars passengers? Surely the human brain must be capable of computation speeds way beyond our wildest dreams in order to carry out the complex tasks that

constitute our everyday lives. In order to understand how the human brain is capable of such Herculean feats it is important that we understand some basic biological foundations.

1.1 Biological Building Blocks

The functioning of the brain involves the flow of electrical impulses through an intricately woven tissue of 100 billion (give or take a few million) basic computational units called neurons. Each neuron consists of three main sections, the Soma (cell body), Axon, and Dendrites.

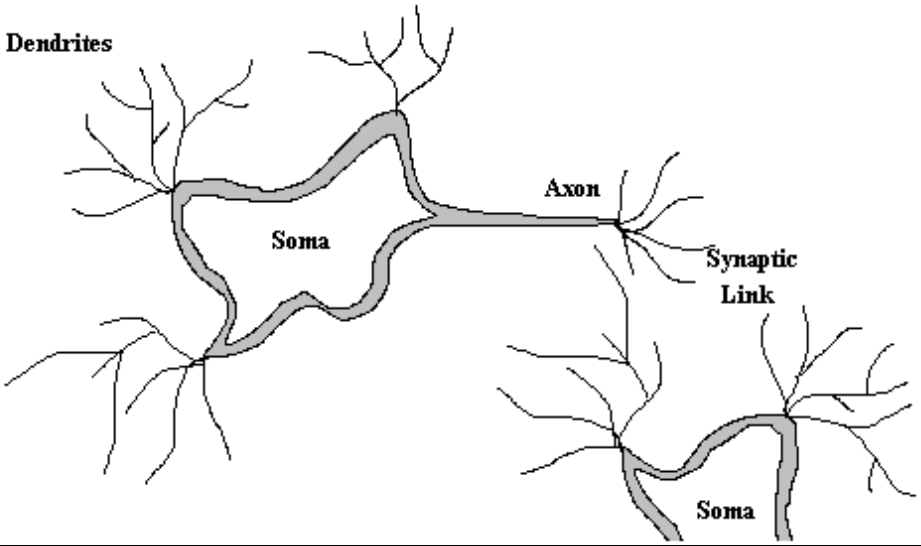

Figure 1.1 Biological Neuron

The soma or cell body ranges from 10 to 100 micrometers in size from which a number of fibrous branches protrude. Each soma has many dendrites that act as 2receptors for incoming electrical signals and one axon that act's as a transmitter. Electronic impulses are transferred between neurons at special connection points called synaptic links. Receiving stimulus or inputs from other neurons through dendrites, the soma or the body of the cell decides when and how to respond. When the cell body receives enough stimulus or input to excite it, it transmits a response to other neurons through the axon. The more a link between two neurons is excited the stronger the link becomes. Each neuron in the human brain may have somewhere in the order of a thousand to 100,000 synapses (connection points). This figure depends on the amount of dendrites a neuron has and its proximity to other neurons. This means that the human brain has as many as 100 trillion synapses making possible the vast and hugely complex computations we carry out through our everyday lives. Although certain areas of the human brain have been attributed with capabilities such as vision [1], speech, and hearing through the study of electrical activity within the brain, individual neuron activity is completely non-deterministic.

1.2 Synaptic Transmission

Neurons communicate through a combination of electrical and chemical reactions. Each neuron is capable of generating an electrical signal that emanates from the soma and then travels down the axon

at about 300 feet per second (200 miles per hour) until it reaches the end of an axon fibre. At this point a chemical reaction is required in order for the electrical impulse to jump across the gap between axon and dendrite (the synaptic cleft). This jump is achieved through molecules called neurotransmitters that travel from one side of the synaptic cleft to the other where they are received by the dendrites of a neighbouring neuron and converted back to an electrical impulse. This electro chemical conversion delays the inter-neuron communication process by a few milliseconds.

1.3 Brain Speed

So if the human thought process operates within a millisecond time frame surely we should be able to recreate it using modern day processors that are capable of crunching more than a million bits per second. The key however is not speed; to the brain a delay in the order of a few milliseconds is negligible because each of its 100 billion neurons can fire simultaneously at about 10 million billion times per second**.** So the computational power of the human brain is not just based on the speed of its connections but on the fact that all of its neurons are working in parallel.

The table below shows a comparison of processor speeds and their comparable brainpower [7].

Processor	**Brain Power**
SUN 4 (200 million bps)	100,000 neurons (Snails Brain)
Cray-3 (100 billion bps)	65 million neurons (Rats Brain)
???	100 billion neurons (Human Brain)

As one can see form the comparisons made above processor speed still has a long way to go in order to match the processing speed of the human brain. Even if we can create a processor comparable to 100 billion neurons it will only be comparable in speed not in function. So where does this leave us?

It seems logical that since the brain has many simple computational units that work in parallel [5] that we should concentrate our efforts not on faster and faster machines but on simple processing units that are interconnected with many other simple processing units.

2. Inherent Parallel Features of Artificial Neural Networks (ANN's)

In order to understand the inherent parallel features [6] of artificial neural networks we must take a brief detour and cover some of the milestones in ANN development [4].

In 1949 Donald Hebb realized that the strengthening of the synaptic link could account for human memory. Hebb proposed that short-term memory was established by a reverbitory circuit in an assembly of neurons and long-term memory was based on some kind of structural change between

neurons based on prolonged activity. From this neurophysiological research Hebb proposed his Learning Rule [2]:

'When an axon of a cell (A) is near enough to excite cell (B) and repeatedly takes part in firing it, some growth process or metabolic change takes place in one or both of the cells. Such that (A)'s efficiency of firing (B) is increased.'

Frank Rosenblatt described the first operational model of neural networks in 1958 by putting together the work of Hebb, McCulloch & Pitts. The perceptron shown in figure 2.1 was based on McCulloch, Pitts (1943) Binary Decision Unit (BDN), it takes a weighted sum of its inputs and sends an output of 1 if the sum is greater then its threshold value, otherwise it outputs 0. The perceptron itself consists of the weights, the summation processor, and an adjustable threshold processor. Figure 2.1 also shows the similarities between the biological neuron and the perceptron

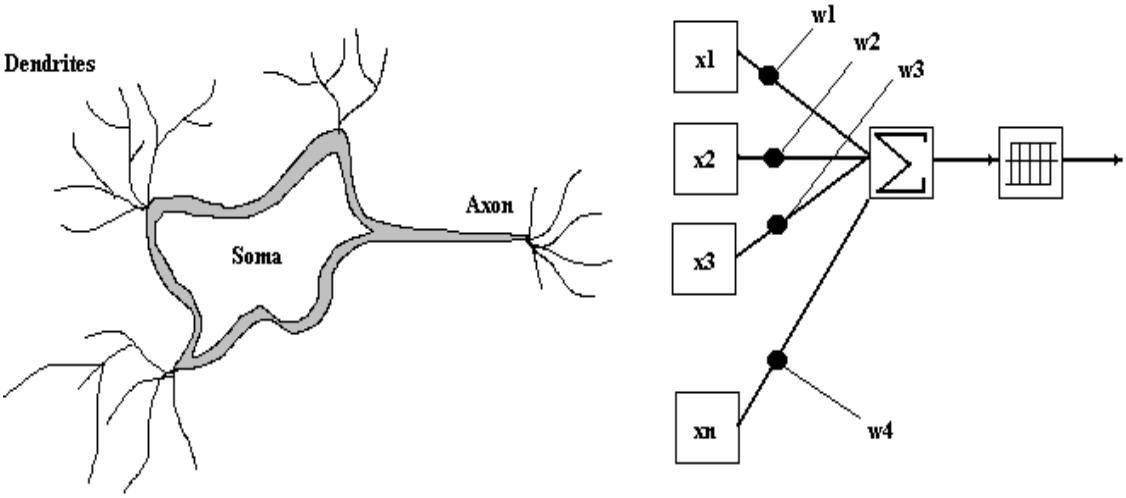

Figure 2.1 A Biological Neuron and a Perceptron

Although the perceptron itself is only capable of solving toy like problems, many perceptrons can be connected together to solve more complex problems. As the network above has only one computational unit there is not much potential for parallel computation.

Consider however the network in figure 2.2, this network has more than one computational unit each of which can operate in parallel.

When values from the input units $(x1-xn)$ are propagated forward through the network each computational unit receives stimuli from every input unit. This means as soon as a computational unit has received a full set of stimuli it can work. As there is no interconnection between units on the same layer each computational unit is completely independent from its adjacent units and can compute in parallel.

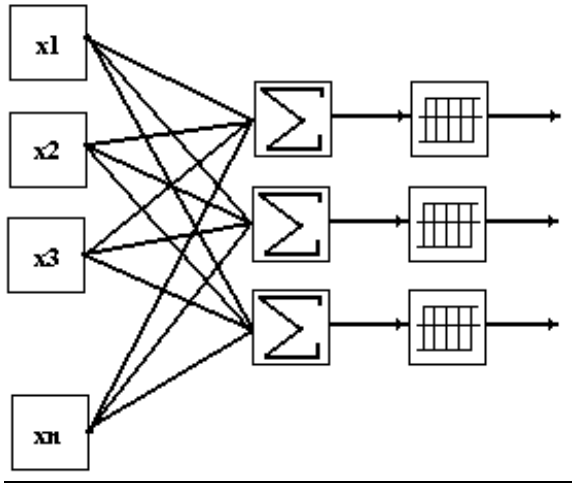

Figure 2.2 A Perceptron with many inputs and outputs

One drawback of the perceptron is that it can only solve problems that are linearly separable. The classification problem shown in figure 2.3 is linearly separable because a line can be drawn to separate both output categories. The line separating the two categories is sometimes called the decision surface.

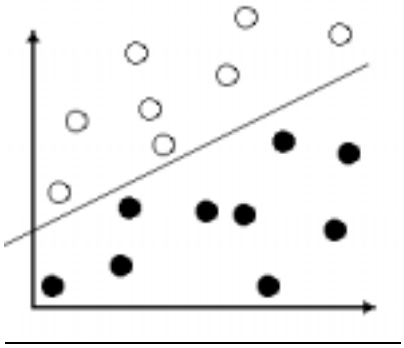

Figure 2.3 Linearly Separable Pattern Classification Problem

These simple computational units are found in all neural networks and their independent behaviour gives rise to a huge potential for parallel computation. Two such networks (ADALINE, and Backpropagation) and their potential for parallel computation are described in the following sections.

2.1 ADALINE Network

The ADALINE network developed by Bernard Widrow (1963) is a simple processing element capable of sorting a set of input pattern into two categories. This sorting task may sound like a trivial

classification problem but the characteristic that sets the ADALINE apart from traditional methods is its ability to learn through a supervised learning process.

The ADALINE is trained by repeatedly presenting it a training set composed of input patterns and their desired outputs. Learning occurs as the ADALINE minimizes the number of errors it makes when sorting the patterns into their correct categories. Once trained the ADALINE can categorize new inputs according to the experience it gained through the learning process. It is this learning phase that constitutes the vast majority of network processing time as running the network in normal operation (actually using its experience) is a simple matter of propagating a set of values through the network and checking the output.

The architecture of the ADALINE is the simplest of all neural networks. As shown in figure 2.4 the ADALINE consists of two layers, an input layer and an output layer. The input layer contains an input unit for each input to the network and a bias unit. The output layer contains only the ADALINE unit which produces the output for the network. Each unit in the input layer is connected to the ADALIINE unit with a link. Each input pattern presented to an ADALINE during training or normal operation should have its components normalized into a predetermined range (typically -1.0 to 1.0). This normalization prevents one component of the input pattern from dominating and possibly interfering with the networks operation. The desired outputs used in the training set should always be one of two values corresponding to the two categories (usually -1.0 and 1.0).

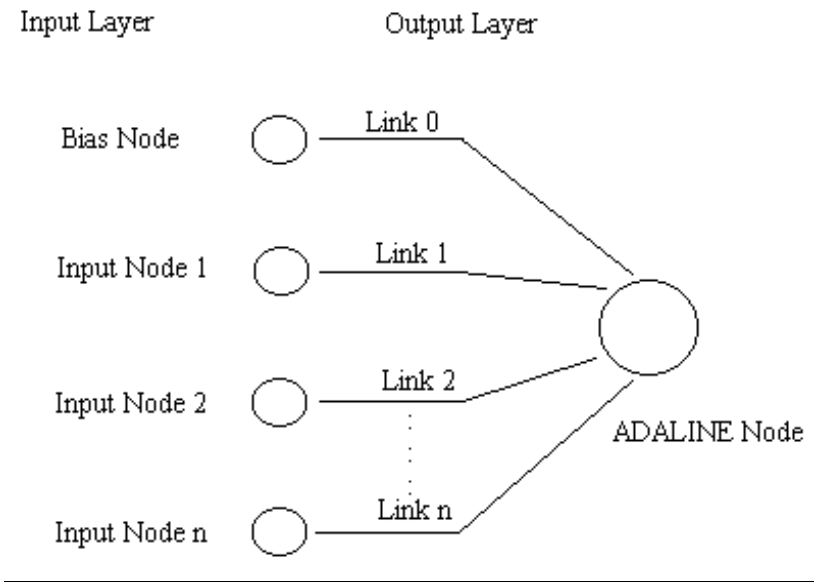

Figure 2.4 ADALINE Network

The ADALINE unit produces only two output values; one corresponding to each of the categories the network can discriminate from. This value is computed by summing the product of each input unit value and its corresponding links value. A threshold function is then applied to the weighted sum

forcing the output value into one of the two categories. As shown in figure 2.5 the threshold function transforms any positive sum into a value associated with category (1.0) and any negative sum into a value associated with category (-1.0).

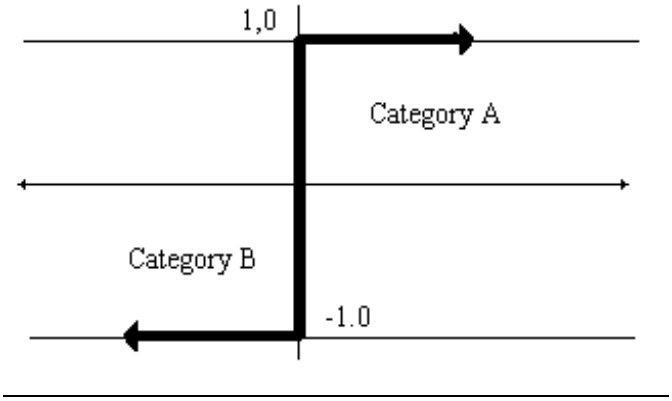

Figure 2.5 ADALINE Threshold function

The ADALINE network learns by modifying the weight values associated with the links during the training process. Training is possible due to an approach Widrow and Hoff (1960) devised to minimize the ADALINE's error by adjusting the link values with a learning law commonly referred to as the delta rule.

Error = Desirder_Output - ADALINE_Output
*New_Weight = Weight + Learning_Rate * Error * Input_Value*

The delta rule uses the error produced by the ADALINE (sorting a pattern into the wrong category) to correct the link values so the correct answer will be produced the next time the input pattern is presented. Since the desired output is either -1 or 1 and the ADALINE's output is either -1 or 1, the error can only be -2 or 2. The delta rule changes the weight values by a percentage called the learning rate (usually around 25-50%). Unfortunately, when the link value is adjusted to give the correct output for the current input pattern, it may cause other input patterns that were producing correct outputs to produce incorrect outputs. The training set must be presented many times before all of the input patterns will produce the desired outputs. If the total error for training set is plotted for all possible link values, the error surface will be a paraboliod as shown in figure 2.6, whose low point is the set of link values that produces the minimum error for each input pattern in the training set. The delta rule traverses the error surface downhill until it reaches the minimum point in the paraboliod. Therefore, the initial link values are not important and can be set to a random value (usually between -1.0 and 1.0) before training begins.

Although the ADALINE works quite well for many applications, it is restricted to a linear problem space. The input patterns in the ADALINE's training set must be linearly separable, otherwise the

ADALINE will never categorize all of the training patterns correctly even when it reaches the low point of the error surface. However compared to other network architectures like backpropagation the ADALINE network is guaranteed to reach its minimum error state since there are no obstacles along the error surface (like local minima) to interfere with the training process.

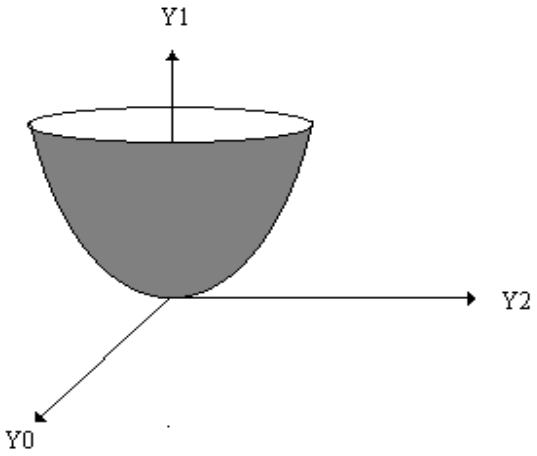

Figure 2.6 Error Surface Produced by ADALINE

The ADALINE network just like the perceptron has only one computation unit thus the potential for concurrent processing is limited. However, more complicated multi-layer networks such as backprop use multiple ADALINE like computation units so a multithreaded ADALINE unit could be utilized by a backprop network. Section 3 of this paper deals with developing a multithreaded version of the ADALINE network.

2.2 BackPropagation Network

The backpropagation (backprop) neural network first formalized by Werbos (1974) and later by Parker (1985), and in 1996 by Rumelhart and McClelland [1] is essentially an extension of the ADALINE network. Instead of having one processing unit, the backprop network is a collection of units (each very similar to the ADALINE unit) organized into interconnected layers. The layered structure of the backprop network allows it to escape the ADALINE's linear separability limitation making it a much more powerful problem-solving tool. Also, the backprop network is not limited to a single binary output, it can have any number of outputs whose values fall within a continuous range. The backprop network is ideal for problems involving classification, projection, interpretation and generalization.

Just as the ADALINE network contains an input and output layer, the backprop network contains these layers as well as one or more middle (hidden) layers. The example in figure 2.7 shows a sample backprop network with one middle layer, two input layer units and two output layer units. The units in the backprop network are interconnected via weighted links, and each unit in a layer is generally connected to each unit in the succeeding layer leaving the output layer unit to provide output for the

network. When an input pattern is presented to the backprop network, each input layer unit is assigned one of the input pattern component values. The units in the next layer receive the input unit values through the links and compute output values of their own to pass to the next layer. This process continues until each output layer unit has produced an output for the network.

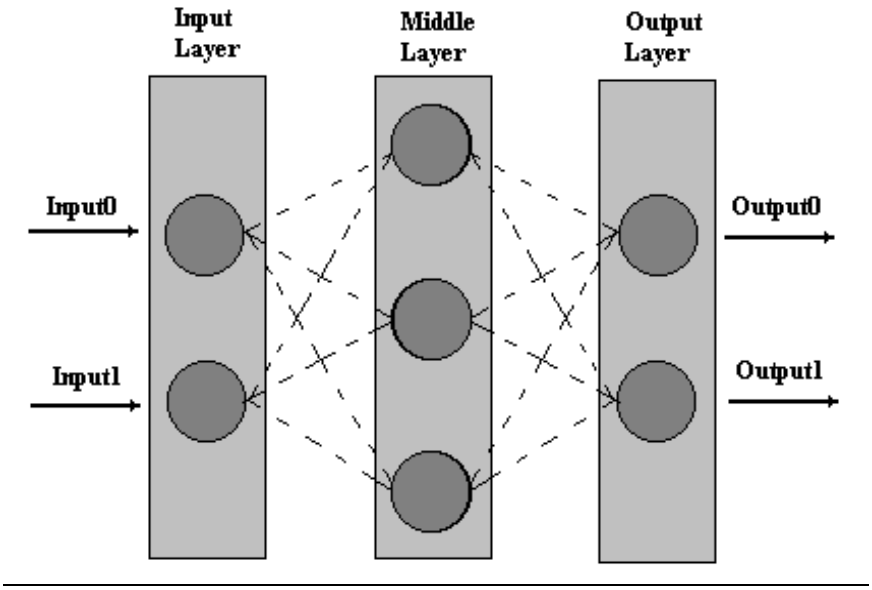

Figure 2.7 Backpropagation Neural Network

Input patterns are identical to the ones used in the ADALINE network except they are usually scaled between 0.0 and 1.0. Since the output layer in a backprop network can have any number of units, a desired output pattern contains a component value for each unit in the output layer. Also, the output layer units are not limited to a binary output like the ADALINE network, theses units (as well as the rest of the units in the backprop network) produce values between 0.0 and 1.0.

After an input pattern has been presented to the backprop network, values propagate forward through the layers until the output layer is reached. The desired output for the input pattern is used to calculate an error value for each unit in the output layer. These error values are propagated backwards through the network as the delta rule is used to adjust the link values to produce a better output. The goal of network training is to reduce the error produced by the patterns in the training set. The training set is presented repeatedly until the overall error is below a given tolerance level. Once training is complete the backprop network is presented with new input patterns and produces an output based on the experience it gained from the learning process.

Unlike the ADALINE network the backprop network is non-linear and produces a more complex surface as shown in figure 2.8. Since the delta rule traverses the error surface downhill, irregularities in the backprop network's error surface may lead to problems (Local Minima).

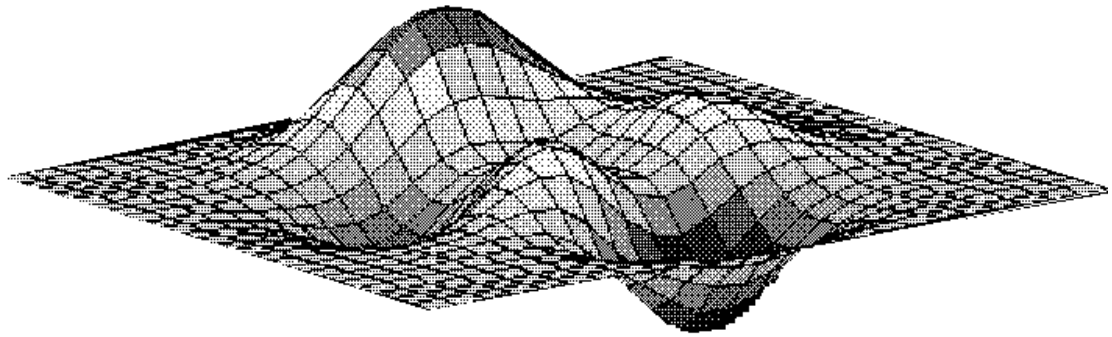

Figure 2.8 Backprop Error surface showing local and global minima

The backprop network holds far greater potential for concurrent processing as it may contain a lot more computational units. Once each unit has received a full set of inputs (stimuli) it can work (sum the weighted inputs and run the result through its threshold function). Because the sequence of processing will depend on the scheduling algorithm used by the operating system that the network is running on, the network will operate in a non-deterministic manner.

3. Multithreaded Neural Networks

One step towards true parallel neural networks is to study how their implementation can be improved by multithreaded/concurrent programming. The approach used here is to treat each network unit as a thread. This means that each unit can operate independently, reading all of its input values, summing them up, and thresholding the result in order to pass it on to all units on the next layer, if any.

As the ADALINE network is simple and is the basis for the more complicated backprop network a simple experiment follows which shows the viability of using multiple threads of execution to represent each unit in the network.

3.1 Defining sequential network operation

The definition of each unit's execution is far easier in sequential implementations of neural networks. In a sequential implementation of the ADALINE network processing would usually begin at the ADALINE unit. This has the effect of summing all of the ADALINE unit's inputs and running them through a threshold function to give some output. This output can be checked against the desired output in order to determine if the network needs to learn. If the network needs to learn the ADALINE unit simply calculates an error using the delta rule and updates all of its incoming weight values. Once the ADALINE unit has finished learning all patterns must be re-presented to the network so the network can finally settle into a stable state (the network is said to have settled once it has successfully learned all patterns). Figure 3.1 shows a typical ADALINE network being run through its paces. The network

is presented with 250 data patterns which it must learn. A complete explanation of the code sample is beyond the scope of this paper but major areas of interest are pointed out.

```
1  while(good < 250)
2  {
3          good = 0;
4          for(i = 0; i < 250; i++)
5          {
6                  Node[0].Set_Value(data[i].In(0));
7                  Node[1].Set_Value(data[i].In(1));
8
9                  Node[3].Run();
10
11                 if (data[i].Out(0) != Node[3].Get_Value())
12                 {
13                         Node[3].Learn();
14                         break;
15                 }
16                 else
17                 {
18                         good++;
19                 }
20         }
21         System.out.println(iteration + ".    " + good + "/250");
22         iteration++;
23
24
25 }
```

Figure 3.1 Typical sequential ADALINE network operation

Line 1 : represents the main loop which will allow the network to continue to run until all patterns are learned, i.e. when the number of good iterations equals 250 (the number of patterns).

Line 2 : initialises a local variable good to zero so that the number of successful iterations can be counted.

Line 4 : starts an inner loop which presents each pattern to the network.

Lines 5 - 6 : load the input units with the next set of patterns.

Line 9 : calls the run method of the ADALINE unit which will begin its computation.

Line 11 : checks to see of the actual output matches the desired output.

Line 13 : invokes the learn method of the ADALINE unit . This calculates the error and updates all the units incoming weight values. Once a pattern has produced an incorrect result all patterns must be re-presented to the network so a break statement forces execution to jump back to the outer loop. Once execution begins again in the outer loop good will be set to zero once more and all patterns will have to be learned again.

Line 18 : simply increment the current value of good as a pattern has given the correct output and there is no need for the network to learn. This keeps execution within the inner loop so that the next pattern will be presented.

Finally when good reaches 250 the outer loop will terminate and the network will have learned all patterns.

The operation of the above network is deterministic, in other words at any given time we can figure out the current point of execution. The operation of the network follows a set of steps, each step depends on the one that went before it. The operation of the network is coded neatly into one main block of code that can be easily followed.

3.2 Defining concurrent network operation

Defining concurrent network operation is far more difficult then one might expect. The best approach is to define the responsibilities of each type of unit within the network and then implement those responsibilities within a thread of execution.

The ADALINE network used as an example here has three input units and one ADALINE unit. Each input unit is connected to the ADALINE unit via a set of weighted links as shown in figure 3.2.

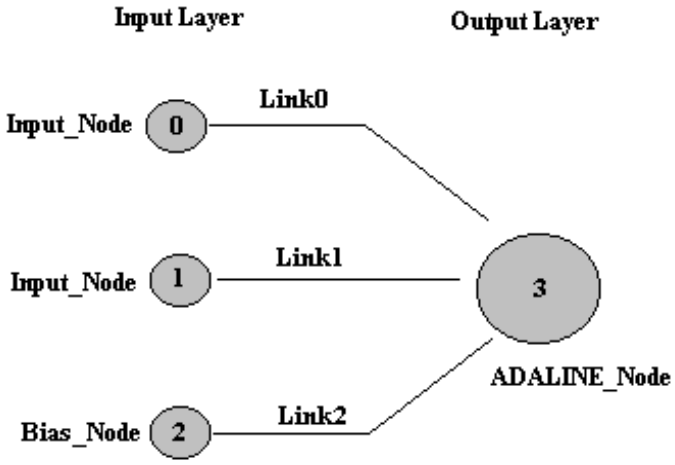

Figure 3.2 Example ADALINE Neural Network

In terms of responsibility within the network the ADALINE unit is the most complex so an obvious place to start is with the input units. Each input unit behaves in the same way, reading values from a pattern file and presenting them to the network. However this time we do not have a main loop with which we can control the presentation of patterns.

3.2.1 Input unit definition

An input unit has a simple set of tasks. Its first task is to take an input pattern and present it to the network. Once it has done this it can lie idle until the next pattern is needed. So when does an input unit know when to get the next pattern? Generally speaking every input unit will have to get its next pattern when the ADALINE unit has finished computing. This means that the input units will have to

wait until the ADALINE unit has received a complete set of input patterns and has finished its responsibilities.

A simplified set of tasks for an input thread is given below:

1. For all 250 data patterns
2. Propagate the input pattern to all units on the next layer, in this case the ADALINE unit.
3. Set the desired result for that particular pattern
4. Update the current pattern number
5. Wait for the single to get the next pattern
6. Check to see of we need to start re-presenting the patterns from the start (an error has occurred on the ADALINE unit).
5. Goto 1

This simplified set of tasks constitutes the run method of the input unit thread. If no errors occur during the presentation of the data patterns then the input units main loop (1) will finally terminate and all patterns will have been presented successfully. Figure 3.3 shows the run method of the input thread.

```
 1  public void run ()
 2  {
 3          for(int j = 0; j < 250; j++)
 4          {
 5                  notify.setRestart(id, false);
 6                  for(int i = 0; i < out_size; i++)
 7                  {
 8                          outlinks[i].write(
 9                          data[current_pattern].In(pattern_component));
10                  }
11                  setExpected(data[current_pattern].Out(0));
12                  current_pattern++;
13
14
15                  try
16                  {
17                          notify.waitToProceed();
18
19                  }
20                  catch(InterruptedException e)
21                  {
22                  }
23
24                  if (notify.getRestart(id))
25                  {
26                          j = 0;
27                          current_pattern = 0;
28                  }
29
30          }
31  }
```

Figure 3.3 Run method of input thread

Line 3 : Represents the main loop for an input unit. When this loop terminates the thread will end its life. The loop is never forced to terminate rather the ADALINE unit notifies the input unit that it should start from the beginning of its input patterns by using a restart notification on line 24.

Line 5 : Once inside the loop restart is set to false for a particular instance of an input unit.

Lines 6- 10: Propagate the input value onto all the units in the next layer.

Lines 11 and 12 : Set the desired output for a particular input pattern and increment the current pattern number so that the next pattern will be read from the file next time around.

Lines 15 – 19 : At this point it is time for the input unit wait until it receives notification from the ADALINE thread. This will stop execution of the input thread until the ADALINE thread is ready for some more input patterns.

Lines 24 – 28 : Once the input thread has been notified by the ADALINE thread it needs to check to see of any errors have occurred. It does this by determining if it needs to restart. If it needs to restart (line 24) then it must set j = 0; and current_pattern = 0;. If this happens the input unit will begin presenting all patterns from the start of the pattern set. If the input unit does not need to start again the j and current_pattern will retain their original values and the next pattern in line will be presented to the network.

NOTE: Once all patterns are learned the loop should terminate of its own accord and the input thread will terminate.

3.2.2 ADALINE unit definition

The ADALINE unit has a much more complicated set of responsibilities for it is the unit that must control the summing of input values, thresholding and learning. An ADALINE unit cannot execute unless it has received a full set of patterns from its input units. This means an ADALINE units does nothing until both input units have propagated their data to the ADALINE unit.

A simplified set of tasks for the ADALINE thread is given below:

1. Continue while the network needs to learn
2. Read and sum all values from the input units
3. Run the summed result through the threshold function
4. Check to see if the actual result matches the desired result
5. If it does then increment good and notify input threads for more input
6. If the outputs do not match then the ADALINE unit must learn by updating all of its input links with the error value. Once the learning has finished good must be set to 0 and the input units must be restarted and notified that they can execute once again.
7. Check to see if all patterns have been learned (good will equal 250)

The set of tasks above constitute the run method of the ADALINE thread. It is important to notice the control the ADALINE thread has over the network operation. It is responsible for waking the input threads up and restarting their pattern presentations from the start if necessary.

Figure 3.4 shows the run method of the ADALINE thread.

```
 1  good = 0;
 2  while(!stopRequested)
 3  {
 4          sum = 0;
 5          for(int i = 0; i < in_size; i++)
 6          {
 7                  double r;
 8
 9                  inlinks[i].setInValue(inlinks[i].read());
10
11                  r = inlinks[i].getInValue() *
12                      inlinks[i].getWeightValue();
13                  sum = sum + r;
14          }
15
16          iterations++;
17
18          sum = transferFunction(sum);
19
20          if (sum != getExpected())
21          {
22                  double error = sum * -2.0;
23                  double delta;
24                  // Must Learn
25                  for(int j = 0; j < in_size; j++)
26                  {
27                          delta = lr * inlinks[j].getInValue() * error;
28
29                          inlinks[j].updateWeightValue(delta);
30                  }
31
32                  good = 0;
33                  notify.restartAll();
34          }
35          else
36          {
37                  good++;
38          }
39
40          if (good == 250)
41                  this.stop();
42
43          System.out.println("Good: " + good +
44          " Iterations : " + iterations);
45
46          notify.proceed();
47  }
```

Figure 3.4 Run method of ADALINE thread

Lines 1-2 : Set the number of good iterations to zero and begin the main thread loop.

Lines 5 – 14 : Loop through all input units and sum their values.

Line 16 : Increment the number of iterations

Line 18: Pass the summed result through the threshold function

Line 20 : Check to see if the actual result matches the desired result

Lines 22 – 33 : An error has occurred on the ADALINE unit so learning must take place. Once all the input links have been updated good = set back to zero and all input threads are notified that they must restart their pattern presentations all over again.
Line 35 – 38 : No error occurred so good is increment and no learning need take place. NOTE: the input threads will continue with the next pattern.
Line 40 : Check to see if good = 250. If this statement evaluates to true then the ADALINE thread is forced to stop. Note the input threads will stop naturally as their main loops will terminate (they will have looped through 250 patterns).
Line 46 : Notify the input threads that they can resume execution.

The above code is a simplified solution to creating a multithreaded neural network. The network has minimum components so as to reduce its complexity. Scaling up to bigger networks would require a lot more code and the addition of new thread responsibilities.

4.0 Conclusion

The multithreaded version of the ADALINE network will not increase the overall speed of the network by much as only one unit is working at any given moment. Execution times for sequential and concurrent networks have not yet been compared so no figures are available to judge speed increases if any. However if the ADALINE thread and input threads are modified to work within a backpropagation network significant increases in speed may be recorded.

4.1 Multithreaded Java Programs and the Java VM

The multithreaded version of the ADALINE network was developed in Java and utilizes Java's input and output pipes as a form of thread communication. One important factor in the next stage of development and testing will be the exploitation of machines that have multiple processors. If the underlying operating systems and the Java VM exploit the use of more than one processor, multithreaded Java programs can achieve true simultaneous thread execution/parallel processing. In theory a Java program would not have to be modified because it already uses threads as if they were running on different processors.

4.2 Extending the Multiprocessor approach

The ADALINE experiment uses Java input and output pipes fro thread communication. This means that threads can only exploit processors within the host machine. By modifying the communication mechanism to use sockets it would be possible for threads to communicate across a network of machines. This greatly increases the number of available processors; it is possible that with a suitable PC network and a suitable neural network that each unit in the network could have its own dedicated

processor which would greatly increase the speed at which the network could learn. There is however other considerations such as the overhead in communications which would have to be addressed but overall the work done so far looks promising.

References

Marr D 1969 A Theory of cerebellar cortex J. Physiol. 202 437-70

D O Hebb 1949 The Organization of Behaviour. Wiley New York.

Rumelhart D E, Hinton G E and Williams R J 1986 Learning Representations by back-propagating errors Nature 323 535-6

Haykin S 1994 Neural networks – a comprehensive foundation, Englewood Cliffs, NJ: Prentice Hall

Kramer A H and Sangiovanmi-Vincentelli A 1989 "Efficient Parallel Learning Algorithms for Neural Networks Advances in Neural Information Processing Systems.

James A Anderson 1986 Cognitive capabilities of a parallel system. In Bienenstock et al.. editors NATO ASI Series. Disordered Systems and Biological Organization, volume F20, Spring Verlag.

Michio Kaku 1998 Visions Oxford.

Bibliography

Michael A.Arbib, editor (1995).The Handbook of Brain Theory and Neural Networks,. The MIT Press

Rumelhart, McClelland, and the PDP Research Group (1996).Parallel Distributed Processing, Volume 1: Foundations,. The MIT Press,

Eric Davalo and PatrickNaim (1993). Neural Networks,.Macmillan computing science series,

Siu , Kai-Yeung, Vwani Roychowdhury, Thomas Kailath (1995). Discrete Neural Computation: A Theoretical Foundation,. Prentice Hall,

Greenfield, Susan (1997).The Human Brain: A Guided Tour,Pheonix,

Pinker, Steven (1997). How the Mind Works. Penguin Science,

Hyde, Paul. (1993.3) Java Thread Programming: The Authorative Solution,.SAMS,